Using Shell Scripts to Parallelize Running Jobs in ABAQUS

Joyce Zhu

August 4, 2014

Introduction and Motive

If you have a long, complex ABAQUS batch job you are submitting to a high-performance computing cluster, you will be able to considerably speed up runtime by running a number of smaller parallel jobs (submitting multiple jobs to the cluster all at once). This entails treating each step of your job as a separate job. The shell scripts and Python data collector script I have written allow for quick parallelization by automatically creating a directory for each job with a copy of each file needed for your simulation; each job has the values of variables changed as you specify. A second script automatically submits all the jobs for you, and the Python data collector script automatically finds the results you want in the .dat file ABAQUS produces and writes them all neatly into one output file which you can then parse later.

Files you should have from the .zip folder:

--prepareAbaqus.sh (creates directories for all jobs and modifies variables in simulation)
--submitAbaqus.sh (submits all jobs)
--runAbaqus.sh (shell script actually submitted to SLURM/SGE/your batch queuing system)
--readAbaqusParallel.py (pulls out relevant data from all .dat files into one central output file)
--two example nonparallel and parallel Python scripts (don't use these in your simulation!)

You should supply:

--Your Python script file for ABAQUS to run
--Any other files needed to run your simulation, like a FORTRAN subroutine
--A (text) file in which you want to collect final output. Note that the Python output collector script will overwrite anything currently written in your file.

Modify your job scripts for parallelization

Any loops you are using to create multiple steps with different variable values should be removed, as the parallelization will instead create multiple one-step jobs with these different variable values. The example below is drawn from my group's work. If you still need help, take a look at the two example nonparallel/before and parallel/after Python scripts provided.

```
CurStep = PreStep
CurJobName = FILENAME

for NN in range(int(1.0/Reso)):
    # Loop GX
    K1 = NN*Reso*pi/AA # Horizontal X-axis
    K2 = 0.0*pi/BB # Vertical Y-axis
    # K3 = 0.0
    STEPNAME ='GX_'+str(NN)
    ##############################################

    ### Frequency Step
    PreStep = CurStep
    CurStep = 'Step-'+STEPNAME
    mdb.models[CurModelName].FrequencyStep(name=CurStep, numEigen=numResult, previous=
        PreStep)

```

Segment of script BEFORE parallelization. Note we are creating multiple steps with different K and step name values.

```
##############################################################################
### Define Step

### Deformation and Post-Buckling Step
CurJobName = FILENAME
mdb.models[CurModelName].FrequencyStep(name='step', numEigen=numResult, previous='Initial')
```

After we modify it, we just assign the step an appropriate name. Where are the variables? We will use the preparation shell script to change the text file itself. The shell script will search for the line which assigns a variable value (eg, find "KX =" in the FORTRAN subroutine file, and replace it with "KX = 0.6").

DESCRIPTION OF EACH FILE

PREPARATION SHELL FILE (prepareAbaqus.sh)

--Parameters changed by script: none! This is the first shell file you run.

--Parameters you must change by hand before running the prepare script: All of them; read below for more details.

Be sure to update the variables at the beginning of the script. The first three lines represent the Python script you're running in ABAQUS, the shell file you want to submit to SLURM (runAbaqus.sh in this case), and your FORTRAN subroutine file (if you have one). The next 3 are MATLAB files we are currently using to process output, so feel free to delete all lines pertaining to them if you'd like.

The dat_reader Python script (readAbaqusParallel.py, in this case) will copy relevant output from the .dat files produced by each ABAQUS job into one final output file. Make sure to go into this Python file to specify the path of this final file (in variable FINAL_OUTPUT_FILE), as this is more efficient than using the prepare script to change the name in every copy of the file.

The "names" variable holds the names of all steps in your experiment. If you'd like to add/delete steps, just follow the syntax (eg a 4th job would add [4]="foo" before the closing parenthesis) and the code will take care of the rest.

The next group of variables pertains to your experiments itself; set your initial Kx/Ky values, your K step size, resolution, total number of jobs, and jobs per step. You can add, delete, or modify these simulation-specific values as you wish. Remember to specify the paths for your home/scratch and group/results directories as well. You shouldn't have to modify current_job, total_jobs, or step_name; these are used to loop through the prepare script itself.

The rest of the code (inside the double for loops) is rather straightforward. We use the sed shell command to search for and replace relevant entries in the other files (mostly Kx and Ky values in the subroutine file and filenames/current step names in the other files). We then make a directory for each job, copy all the modified files into the folder, and update variables as necessary.

You can modify the sed commands to search for and replace different lines/elements in your own scripts. Most modifications will just involve replacing the text which you are searching for (eg, changing “FILENAME =” to “step_name=” if you have different variable names), but for more complex substitutions you may need some more detailed knowledge of the sed command line tool and regex. See the included crash course to shell scripting if you’d like help.

MASS SUBMISSION SHELL FILE (submitAbaqus.sh)
--parameters changed by shell: none, as this just uses a loop to submit all scripts
--parameters you must change by hand before running the prepare script: Job_limit, if you don't want more than X number of jobs in your queue; any variables you modified in the prepareAbaqus.sh file MUST match up! I'll explain the start and end jobs variables below.

This script uses a loop to change into the directory for each job, submit the job's shell script (run_abaqus.sh) to SLURM, then go back up a directory and repeat the process. As mentioned above, you can set a job limit if you don't want more than a certain number of jobs in your queue. The script will sleep until the number of jobs dips below the limit, after which it resumes submitting the next job. If having a queue limit doesn't matter to you, just comment out the "sleep 30" line.

Another nice feature is that you can modify which jobs you want to start and end with when submitting -- it's currently set to the default of jobs 1 to the end, aka all of them. This is particularly useful in the case that you want to re-run only a particular subset of jobs. For example, if you're running 60 jobs and want to rerun jobs 20 to 40, you can just change those variables instead of manually submitting the 20 scripts because you don't want to run all 60 again.

If you do NOT use SLURM for job submission, you'll have to modify the lines including sbatch and squeue. Sbatch is the command to submit a job to SLURM (the equivalent is qsub in SGE), and squeue –u $USER checks the jobs you are running (the equivalent is qstat in SGE).

INDIVIDUAL SHELL FILE (run_abaqus.sh)
--parameters changed by shell: names of script/subroutine filenames to call, name of job/directory
--parameters you must change by hand: any SLURM (SBATCH) options you want to tweak; make sure any variables shared between this script and the prepare script, like the name of the Python script file, match up

This shell file is the one submitted to the queue for each job you'd like to be run. It loads needed packages (I've loaded Intel FORTRAN Compiler 11.1, ABAQUS 6.12-1, and GCC; feel free to remove/add your own), then runs ABAQUS with no GUI using your Python script. Once ABAQUS is done processing, the script calls the Python script which reads relevant output from the .dat file to a final output file (see below), then copies any important files into your results/group directory.

Don't worry too much about the exit gracefully function -- that just allows for termination of the job when necessary.

If the "module load" commands are unfamiliar to you, that's how our group gets access to needed software on our server. You can delete these lines if you don't need to load software; if you have different commands to do so, modify the code accordingly.

You will need to modify the specification of job options (in SBATCH, these are represented by the #SBATCH lines) in this script if you are using a different job submission engine (eg, SGE/qsub instead of SLURM/sbatch). If you are running a .inp file and not a Python script in ABAQUS, modify the "abaqus cae" line as necessary.

ABAQUS .DAT FILE READER (readAbaqusParallel.py)
--parameters changed by shell: the name of the .dat file to be read, the current step/job we are on
--parameters you must change by hand before running the prepare script: path of your output file

The obvious problem with running a number of jobs in parallel is that your results are now scattered over all the separate job directories. You could modify the runAbaqus.sh script to, instead of making a subdirectory for each job in the results folder, copy relevant output files all to the same directory. However, in this case the output is STILL spread all over your output files. What we want is to have ALL output from all your jobs neatly written to a single output file.

That's where this Python script comes in. A copy is placed in the Result subdirectory of each job's directory and is called by a runAbaqus.sh file after ABAQUS has finished processing. The output file path is the same in each copy of the script, so each time the script is called, it will look through its corresponding .dat file for the line containing "(RAD/TIME)" (or whatever else you'd like). Once it has located this line, it copies all relevant data into an array, then rewrites the existing output file to include the data from this most recent step.

The result in your central output file is easy-to-parse results. Each column represents the data from one step, and each column is separated from the next one by a tab. (However, there are obviously no tabs before the first column – remember to account for this when you write any code to parse the file!)

You can modify the script to pick out and write whatever types of data you want; the comments will explain what each section of code is doing. Basic knowledge in Python is required.

Once all the jobs have finished, the Python scripts have copied all relevant outputs from the .dat files into your specified final output file. Now you can use whatever tools you'd like (shell sed/grep/awk, another Python script, MATLAB, etc.) to parse the results of this final text file!